# Evaluation of Dynamic Compilation Technologies for Cognitive Information Processing Architectures

**Final Report**

PI: Sudhakar Yalamanchili

School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA. 30332
sudha@ece.gatech.edu

# 1 Introduction

The COGENT architecture represents a unique execution model for the implementation of cognitive algorithms. Its uniqueness stems from the great deal of flexibility it offers to orchestrating the execution of highly data dependent parallel algorithms. In addition to supporting models such as the Bulk Synchronous Parallelism (BSP) model, it naturally supports flexible extensions that permit asynchronous barriers. In general, cognitive behavior that is based on inference relies on the creation, manipulation, and otherwise processing of a very large number of relationships which in COGENT are captured in graphs. The anticipated size of the knowledge base is extremely large whereas the anticipated computations that execute over the knowledge base will by comparison access a relatively small percentage of the knowledge base. Moreover an embedded cognitive application may be processing thousands of such computations or *queries* at a time leading to a novel dynamic execution model wherein queries are dynamically invoked at the location of the data. Code may be pre-placed or dynamically compiled and loaded at run-time in a demand-driven manner. In either case, algorithms formulations can rely on the ability of all processors to "effectively" share code in the same sense of a shared memory parallel machine.

This section describes our experiences with mapping parallel implementations of some graph algorithms to the COGENT architecture. We present performance results from the COGENT simulator as well as summarization of the lessons learned and consequent opportunities in future instantiations of this architecture.

# 2 Parallel Graph Algorithms

The graphs contained in the knowledge base capture relationships between physical entities, events, observations, etc. Structural relationships between graph vertices can correspond to important inferences about the real world. Among the various such structural relationships is the existence of a path between two vertices and the set of vertices that can be reached from a vertex. Both of these properties can be generalized to analyze attributes of the edges and nodes along the paths. Therefore, we started with an analysis of the performance of algorithms for the shortest path and vertex reachability. The execution model is one wherein a query requests the shortest path between vertices or the set of vertices reachable from a specific node, respectively. Multiple independent queries can be executed concurrently on the machine. A sequence of steps in the concurrent computation of two queries for reachability analysis is illustrated in Figure 1.

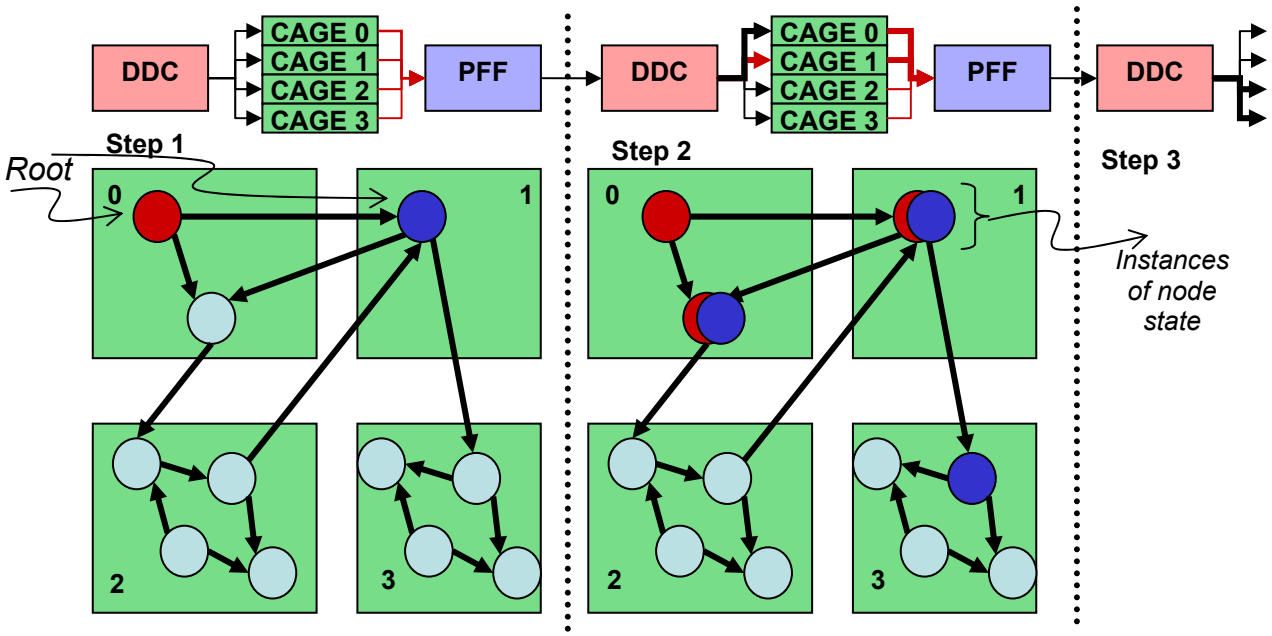



**Figure 1 Execution of Multiple Query Instances**

## 2.1 Node Reachability

Reachability analysis computes the set of nodes that can be reached from a specific node via paths in the knowledge base (represented as a directed or undirected graph). Our implementation utilizes a spreading activation model. The key idea is to perform a Breadth First Traversal of the graph in parallel. Starting form the start vertex, logically a marker is propagated along all outgoing edges to adjacent nodes. Each adjacent vertex is marked and propagates the markers along each vertex's outgoing edges. When an outgoing edge points to a remote vertex, i.e., located on anther CAGE node, a parcel is constructed, where the parcel contains the list of nodes included in the reachability tree to that point. This parcel is transmitted to the remote CAGE node that contains the remote vertex. This remote CAGE node, upon receiving the parcel, propagates the markers among its vertices. If another remote vertex is encountered, a new parcel is constructed, populated with the reachable vertices traversed to that point and the process is repeated. When no further propagation is possible at a CAGE node, a parcel signaling completion is transmitted to the PFF along with the reachability set.

Each query operates in a different context and thus operates in a different thread on a CAGE node. The use of contexts enables the correct, and concurrent isolation and aggregation of partial results for each query. The multiple contexts were not modeled in the simulation analysis. However the extension is straightforward. All local data stuctures created for processing a query are now indexd buy the context. The pseudo code for the reachability algorithm is provided in Section 5.1.

## 2.2 Shortest Path

Our experiments utilized Dikjstra's algorithm for computing the shortest path. The pseudo code for the algorithm is provided in Section 5.2.

## 2.3 Sub-graph Matching

The informal description of the subgraph match problem is the search for the presence of an input graph in a host graph, i.e., the existence of instances of the input graph as a subgraph of the host graph. This is a well known, widely considered computationally intractable problem. However the discovery of specific sub-structures (here equivalently subsets of relationships) in larger graphs has important consequences for cognition. We have developed a heuristic for solving a precise formulation of this problem- the sub-graph isomorphism. The essential structure of the algorithms is as follows.

1. Initialize: The input graph is transmitted to all CAGE nodes which then repeatedly execute the following two phases.
2. Execute Local subgraph matching algorithm:
    a. If SUCCESS, update PFF with the result and SUCCESS status
    b. If partial match, construct new subgraph match request for non-matching component of subgraph and propagate to target CAGE via PFF
    c. Remote vertices are used to identify the possible CAGE nodes on which the remainder of the input subgraph would reside
    d. If no match report FAILURE to PFF
3. Iterate over step 2 until no more parcel requests for subgraph match.

The local subgraph matching algorithm employs a heuristic which orders the vertices to reduce the expected execution time of the algorithm.

1. Sort vertices in non-increasing order of vertex degree
2. Find a candidate vertex for (exact) matching based on vertex degree
3. If adjacency is maintained, match the target vertex with the vertex in the input subgraph and go to step 2 for next vertex
    - Adjacency definition: any mapped neighbors in the input graph are neighbors in the host graph
4. If no match is possible, backtrack to previous match
    - Go to step 2 to find a new mapping for a previously mapped vertex, i.e., un-map a vertex

A more detailed pseudo code description can be found in Section 5.3.
One of the many insights from the construction of the preceding algorithm is illustrated in Figure 2. One observation is that if the input graph exists in the host graph, but is resident across two CAGE nodes 1 and 2, then both CAGE nodes will send additional requests for partial matches to each other as illustrated in Figure 2. Thus there is significant redundancy in the computation. Effectively if a subgraph exists in the host graph and is spread over K CAGE nodes, the subgraph will be matched at least *K* times and at least *K* times as many parcels as necessary will be sent. A CAGE node may receive multiple requests for the same partial match from multiple sources.

However the unique presence of the PFF can be used to filter and prune redundant computations. This in fact suggests a programming model wherein algorithms are structured as the iterative application of purely local computations. This is a simple model that often simplifies algorithm formulation, however leads to significant redundancy. The PFF (can be) is used as the means to exercise global knowledge to prune and eliminate redundant computations.

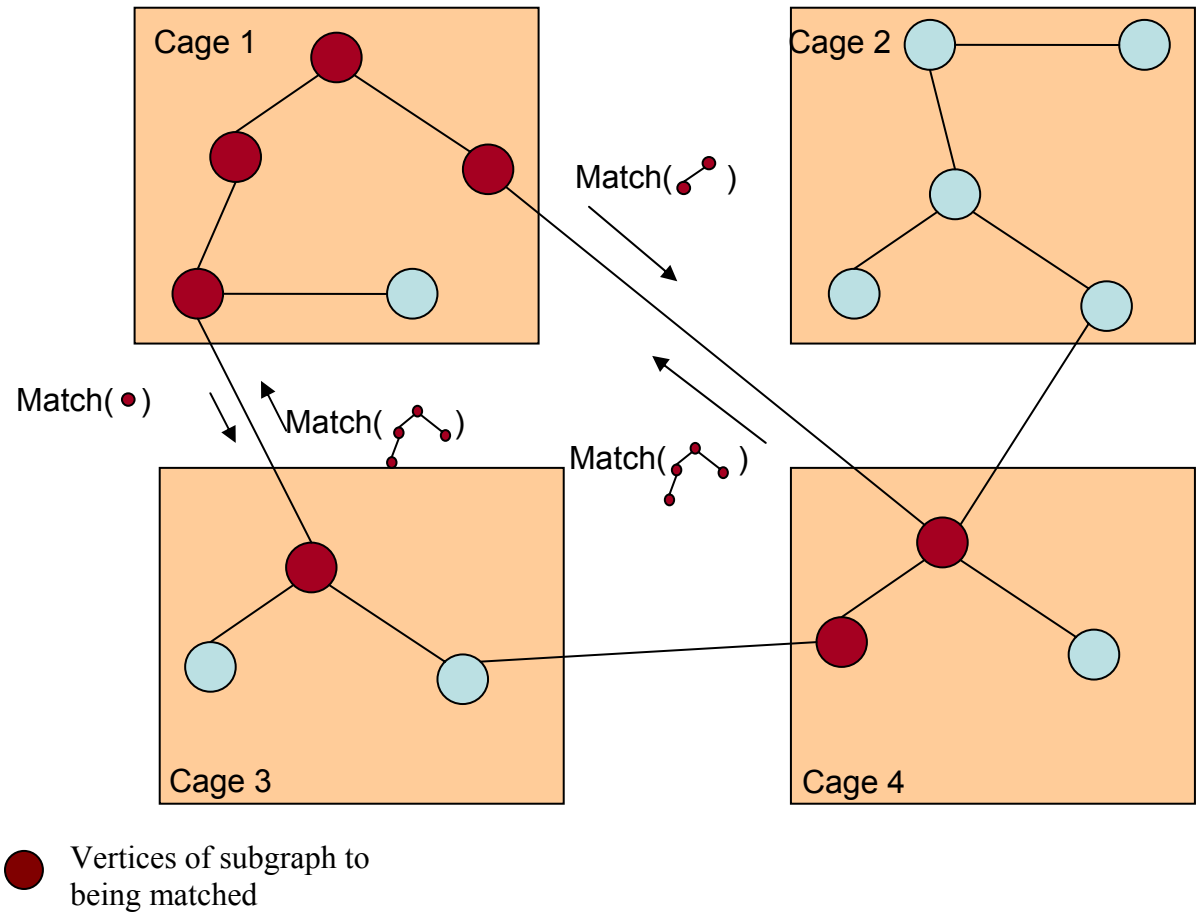



**Figure 2 An Example of Subgraph Isomorphism**

# 3 Programming and Execution Models

The experience with the formulation of the subgraph match algorithm suggests simple extensions to Bulk Synchronous Parallelism (BSP) execution model as the basis for programming the COGENT architecture. BSP is a parallel programming model that abstracts the execution of programs as a sequence of interleaved computation and communication steps. In a computation step, processors execute independent local computations. In the next communication step processors exchange messages and perform barrier synchronization. This simplified model has had success in implementation of a variety of parallel algorithms including graph algorithms. The COGENT architecture suggests a simple yet powerful extension to the BSP model – the idea of *active barriers*. Active barriers are those where functions can be associated with the barrier to process the contents of messages exchanged in the course of the execution of the barrier. These functions can filter messages, may merge messages and may even generate new messages. The use of such a model enables one to employ simpler search algorithms at the expense of the implementation of active barriers in the PFFs.

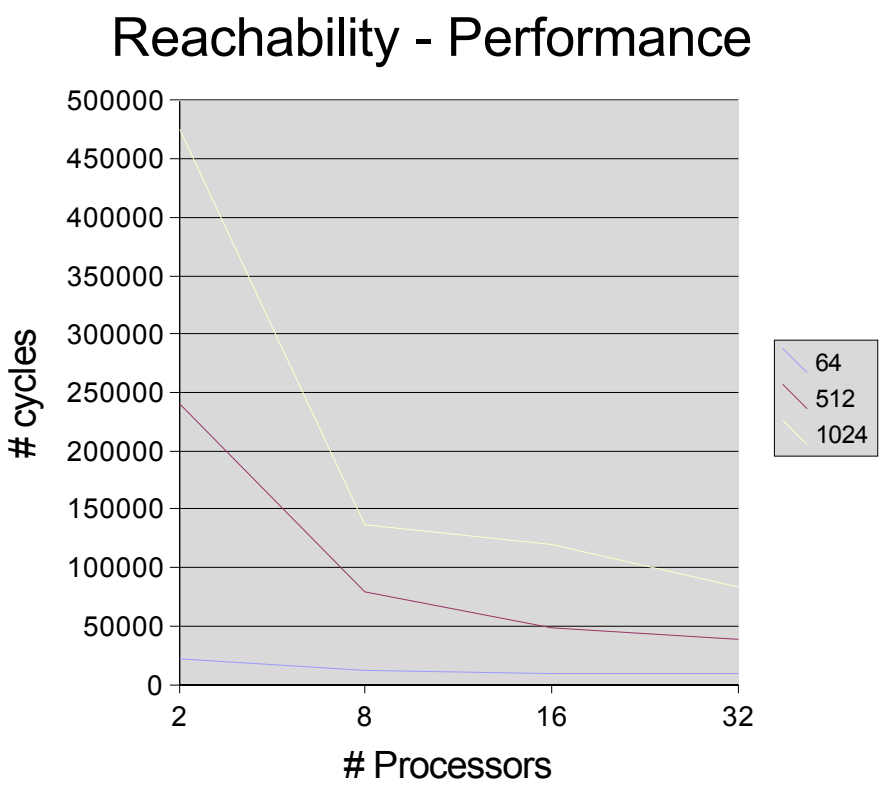
# 4 Lessons Learned

## 4.1 Performance Results

The performance experiments are summarized in the following figures. These figures plot the execution times as measured by the COGENT simulator for graphs of varying size and systems with a varying number of CAGE nodes. Figure 3 and Figure 4 record the execution time and parcel traffic for parallel reachability. The implementation utilized test cases where the host graph was a tree. Such a case reflects a lower bound on the number of parcels one can expect to transmit in a similar sized (number of vertices) graph. The execution time scales well with the number of nodes and appears to be sensitive to the number of vertices on a CAGE node. For the same reason the communication traffic falls off as the number of nodes on a CAGE node drops. From Figure 3 there appears to be an effective number of CAGE nodes at which the amount of parcel traffic and the amount of work done in a CAGE node are balanced. However for graphs with a large number of nodes, e.g., Figure 4, this does not appear to be the case. We conjecture that with a large number of nodes, the base volume of parcel traffic masks any potential increase in the number of parcels for small numbers of CAGE nodes.

While similar execution time behavior can be observed for the shortest path algorithm the parcel traffic is more problematic – it rises considerably as the number of CAGE nodes increase. This is primarily due to the fact that Dijkstra's algorithm does not admit to as much parallelism as in the case of the parallel implementation of reachability analysis. The dominant source of speedup in applications in this case is expect to be across queries.

Preliminary results for a parallel implementation for sub-graph isomorphism are shown in Figure 6 - the execution time as a function of the number of instances of the target graph on the host graph. The host graph used for testing sub-graph vertices was of size 1000 vertices and the sub-graph of size 5 vertices, while the number of CAGES was 10. The figure records the execution time as a function of 2, 4, 6 and 8 instances of the target graph present in the host graph. In these experiments all instances were correctly detected, although in experiments with the presence of a larger number of instances all instances were not always detected by the heuristic. For example, in case of 8 instances, only 6 instances were discovered. The performance and behavior of the heuristic is sensitive to the partitioning of the knowledge base across CAGE nodes and distribution of target graph nodes across the CAGE nodes. While our experiments did not report any false positives, a formal proof of this attribute of the heuristic remains to be done. We note from the figure that the number of instances does not increase the execution time or parcel count substantively. One interpretation is that the base communication load and computation load is quite high and the coverage of the host graph is always extensive. This recording additional instances does not add that much by way of computation or communication. This further suggests that for large knowledge bases, this approach can be quite effective.



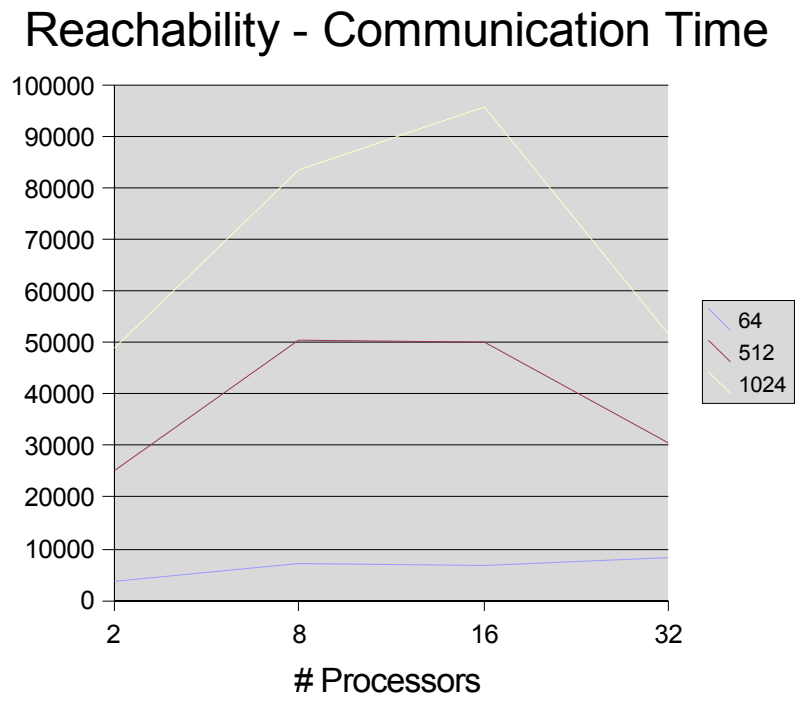




**Figure 3 Results for Parallel Reachability Computation**

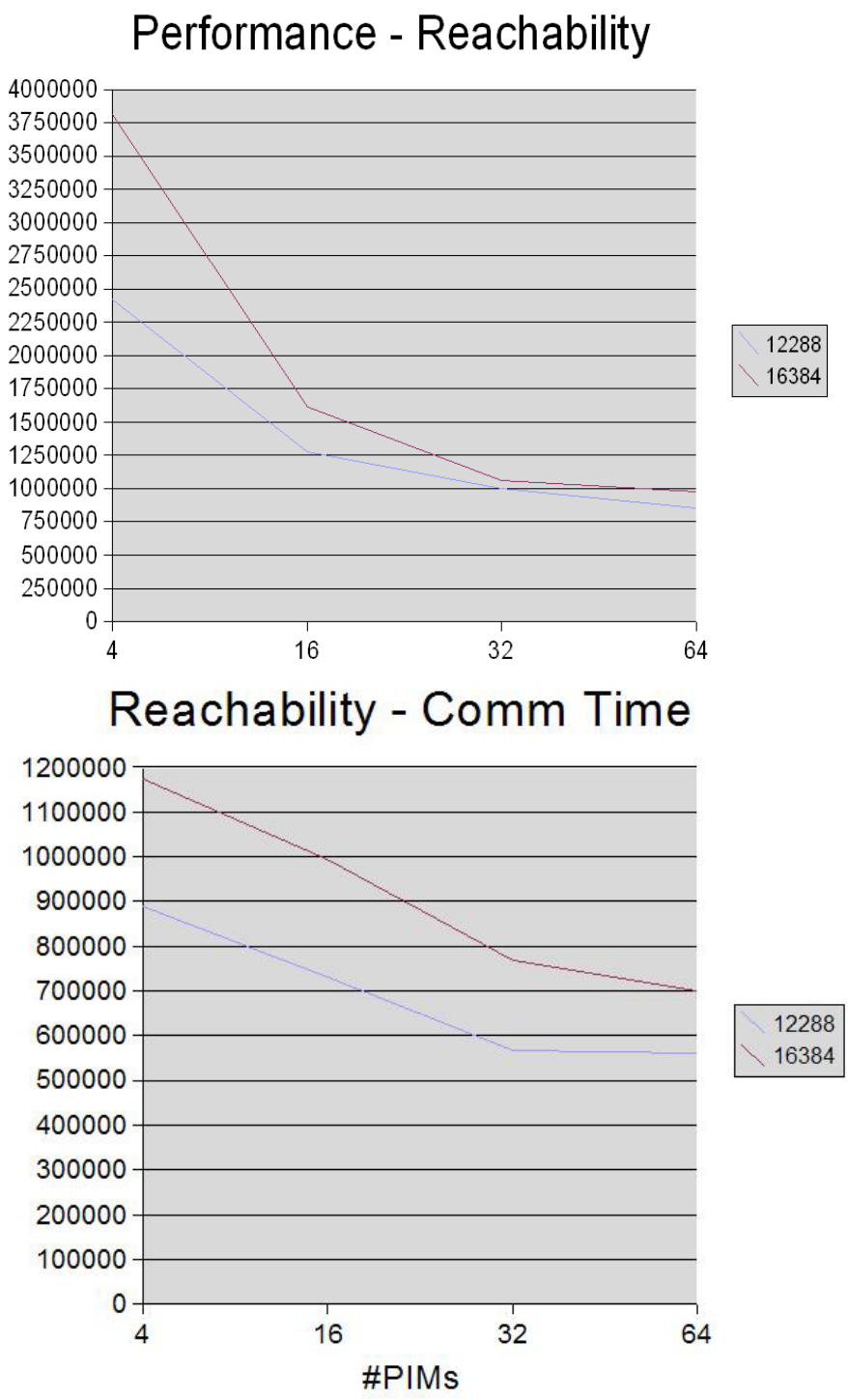
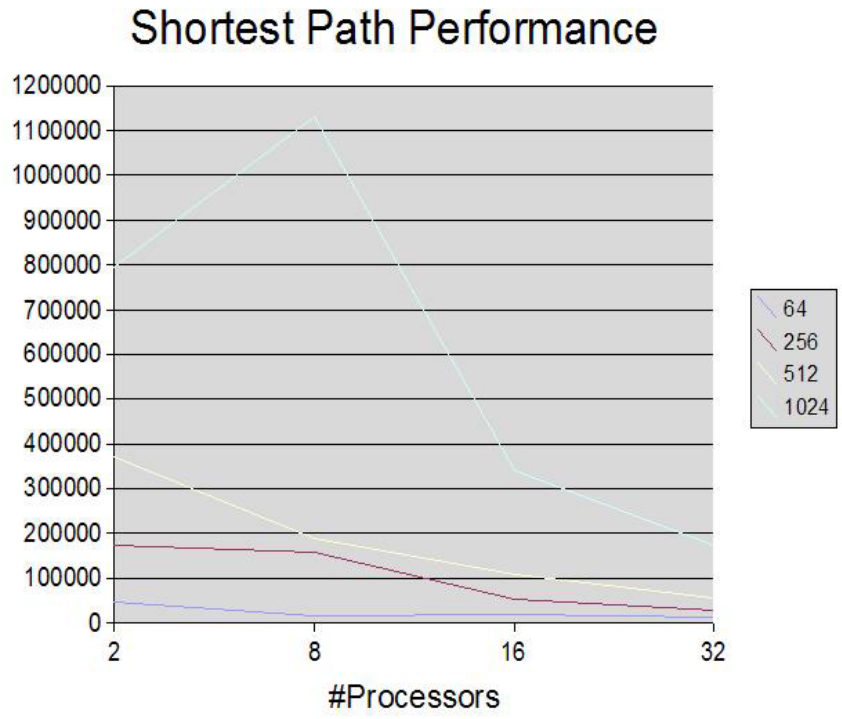



**Figure 4 Results for Parallel Reachability Computation: Large Graphs**



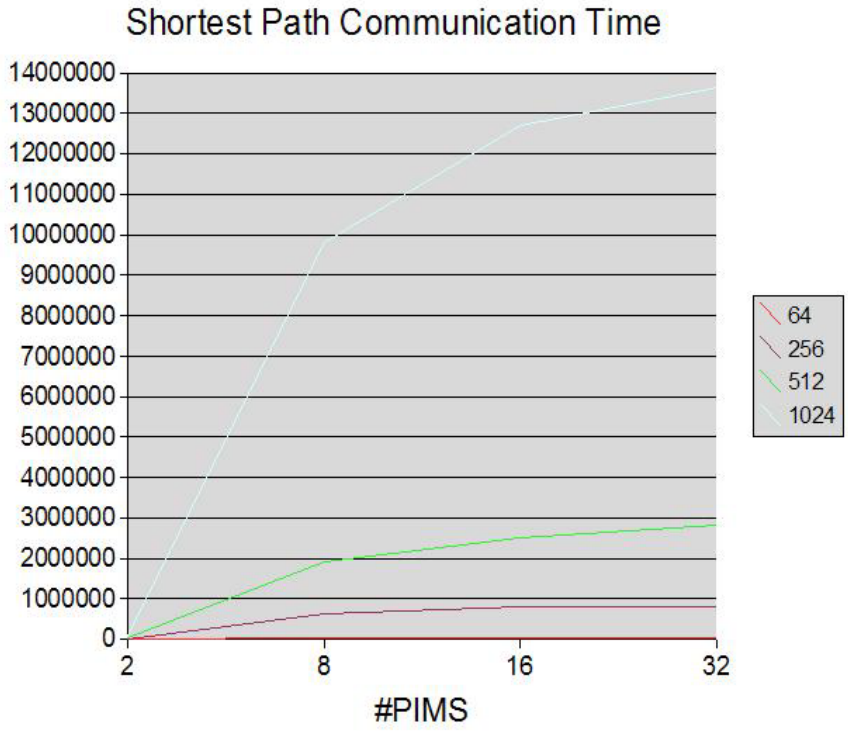
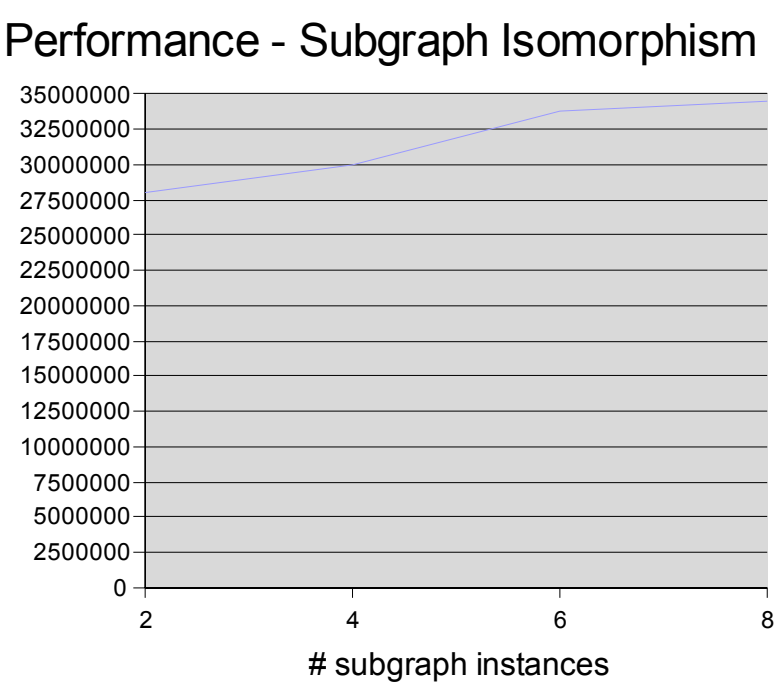




**Figure 5 Results for Parallel Shortest Path Computation**



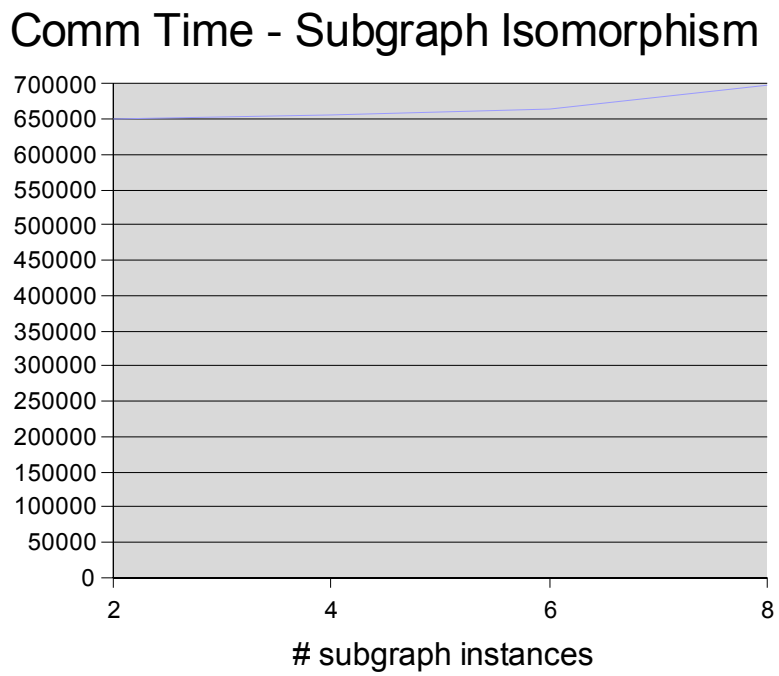




**Figure 6 Results for Sub-graph Isomorphism Computation**

## 4.2 Algorithmic and Programming Issues

The following summarizes additional algorithmic and programming insights.

1. The performance of the subgraph isomorphism algorithm can be improved by employing the following heuristic. Start with looking for a match for the input sub-graph vertex with highest degree. Further, in the target graph order vertices for matching which have degree greater than or equal to that of the sub-graph vertex. Such orderings will narrow the search. Also, in the process of mapping sub-graph vertices to target graph vertices, if a point is reached where the adjacency is not preserved, then the backtracking approach is used to check for other possible assignment.

2. If the sub-graph is spread across many processors, then combining the all possible partial results in the PFF that are computed by individual CAGE nodes would involve substantial number of permutations and combinations.

3. SUGGESTION 1: combining partial results: Suppose Processor P1 computes a partial result and detects that the remainder of the sub-graph lies on processors P2 and P3. It can tag the input parcel with its identity and send a request to P2 and P3. P2 and P3 might compute many possible partial results in continuation with the one computed by P1. Then they would communicate to PFF these partial results and include their identity and matching number in the communication parcel sent to PFF. PFF might then scan through these parcels and detect one or many instances of sub-graph isomorphism.

4. The complexity of combining partial results in the PFF is a function of the number of CAGE nodes that host each matching sub-graph in the host graph.

5. A heuristic can be utilized to minimize the redundant computations generated in as follows.
   If the processor locally computes a partial match of the sub-graph (i.e., without receiving any request from some other processor) and the partial match is of very small size as compared to the size of the sub-graph, then it makes sense for that processor not to initiate any partial match requests, but rather to wait for the request from some other processor. This, is because, the former case would give rise to many partial results. And waiting would result in reduction of redundant computations as well because this processor would receive a request with high probability from another CAGE node. Here, one need to decide on what partial result size should serve the purpose e.g. one third of the sub-graph size.

6. SPECIAL CASE: DIRECTED GRAPH
   In case of directed graph, the matching process might come to halt if a vertex with no out-degree is reached. Some approach needs to be defined to overcome this situation.
   SUGGESTION: send request to its adjacent vertices on other processors and check for adjacency there.

7. SUGGESTION: GRAPH PARTITIONING: Depending upon the graph partitioning algorithm, the number of partial results computed and to be combined varies. Hence, thought can be given to using graph-partitioning algorithms which would reduce this required computation.

8. Testing and Validation: An important problem was the testing of the implementation. As with most computationally intractable problems, it is difficult to generate demanding test cases. Since the host graphs are rather large to begin with how does one construct the test cases? One approach is to extract sub-graphs from the host graph. A second approach is to construct large host graphs by first replicating the test graph and then creating a large host graph by the random addition of edges and nodes to this host graph. We can then assert to the existence of at least the initial number of matches.

9. Our experience suggests a node architecture where special purpose graph processing accelerators are tightly coupled to a threaded processor and local memory. One can prototype such a capability using COTS processors that provide dual core processors tightly coupled to FPGAs which in turn are tightly coupled to large DRAM.

10. SUGGESTION: REMOTE VERTICES: Our experience suggests that the implementation becomes easier if the remote vertices' adjacency is maintained in their dummy nodes. This gives some idea of the structure of the remote graph. Rather than postponing the detection of adjacency to remote CAGE, failure can be determined locally and the number of parcels can be reduced.

11. SUGGESTION: RACE CONDITION: Since, the PCAM sends the requests to CAGEs and CAGEs notify completion to PFF, this may lead to a RACE condition. It would be a better idea to have a central authority which would keep track of current requests sent and completed.

12. SUGGESTION: The PFF may examine the communication messages and selectively prune/modify them. This would reduce redundant parcels and may help combining data and process it.

# 5 Algorithm Descriptions

## 5.1 Algorithm for finding Set of Reachable Vertices from a given source vertex

```
findConnectedComp(SimPIM pim, SimParcel inputparcel) {
   comp = SET of vertices fetched from inputparcel for which reachability is to be computed

   FOR each vertex v in comp do {
      IF (v is a local vertex) {
         remove v from comp
         add v to the SET locallyConnectedVertices

         FOR each vertex u in locallyConnectedVertices do {
            IF (u is untagged) {
```

```
                FOR each vertex w incident to u {
                    IF (w is not remote) {
                        IF (w is untagged AND w is not in locallyConnectedVertices){
                            Add w to locallyConnectedVertices set
                        }
                        Add w to input parcel's partial result
                    }
                    ELSE {
                        IF (w is untagged AND w is not in comp) {
                                Tag w
                                Add w to the SET remoteVerteicesInComp
                        }
                    }
                }//end of FOR
                TAG u
            }
        }
    }
}//FOR each vertex v in comp do

Add vertices in the SET remoteVerticesInComp to comp
Send request to all PIM CAGES on which the vertices in comp are stored locally
}
```

## 5.2 Algorithm for finding Shortest Path from a given source vertex

```
public void findShortestPath(SimPIM pim, SimParcel inputParcel){

    boolean isPimMin = true , isPimMinLess = false;
    boolean isCompMin = true;
    Vertex pimMin = null , remoteMin = null, compMin = null;
    Vector remoteUuids = new Vector();
    boolean compMinFound = false ;

    Vertex graphVertex = null;
    String startName = null;
    int compMinDist = 10000, pimMinDist = 10000, remoteMinDist = 10000;

  comp = SET of vertices in inputParcel

  IF (comp contains no elements) {
      //send message to EM .. result false
      RETURN
  }
```

```
FOR each remote vertex r in Graph do {
    IF (r.distance < 0){
        r.distance = INFINITY
        //this is required because remote vertices are assigned -1 value initially (may be the graph
partitioning)
    }
    IF (comp contains r) {
        temp = vertex in comp
        IF (temp.distance > r.distance) {
            temp.distance = r.distance
        }
        ELSE {
            r.distance = temp.distance
        }
    }
}// FOR each remote vertex v in Graph do

FOR each local vertex v in Graph do {
    IF (comp contains v) {
        temp = vertex in comp
        IF (temp.distance > v.distance){
            temp.distance = v.distance
        }
        ELSE{
            v.distance = temp.distance
        }
    }
}// FOR each local vertex v in Graph do

compMin = vertex in comp with minimum distance
compMinDist = compMin.distance
pimMin =  untagged local vertex in graph with minimum distance

IF (all vertices in graph are tagged) {
    Remove all local vertices in comp
    IF (comp is not empty){
        compMin = vertex in comp with minimum distance
        SEND request to PIM CAGE on which vertex compMin is stored locally
    }
    //SEND sendParcel(pim, new Boolean(false), inputParcel.getContext());
    RETURN
}

pimMinDist = pimMin.distance
remoteMin = untagged remote vertex in graph with minimum distance
```

```
IF (remoteMin is not NULL){
    remoteMinDist = remoteMin.distance
}

FOR each remote vertex r in graph do {
    IF (compMinDist <= pimMinDist) {
        IF ( r.distance < compMinDist){
            Tag vertex r
        }
    } ELSE {
        IF (r.distance < pimMinDist) {
            Tag vertex r
        }
    }
}// FOR each remote vertex r in graph do

IF (pimMinDist <= compMinDist){
    sourceVertex = pimMin
}
ELSE {
    FOR each local vertex v in graph {
        IF (v equals to compMin) {
            sourceVertex = v
            Remove v from comp
            IF (v is untagged) {
                sourceVertex.distance = min(v.distance and compMin.distance)
                BREAK
            }
            ELSE {
                sourceVertex = NULL
                IF (comp is not empty) {
                    compMin = vertex in comp with minimum distance
                    compMinDist = compMin.distance
                    IF (compMinDist > pimMinDist) {
                        sourceVertex = pimMin
                        isPimMinLess = TRUE
                        BREAK
                    }
                }
                ELSE {
                     BREAK
                }
            }
        }//IF (v equals to compMin)
    }//FOR
}
```

```
IF (compMin found in local graph AND compMin tagged) {
    IF ((pimMinDist >= remoteMinDist) || (remoteMinDist <= compMinDist)){
        FOR each remote and local vertex v in Graph do {
            IF (v.distance <= pimMinDist) {
                Add v to comp
            }
        }
    }
}

WHILE ( (sourceVertex is not NULL)  AND (sourceVertex is not tagged ) ) {
    tag graphVertex
    IF (sourceVertex exists in comp){
        Remove sourceVertex in comp
    }
    RELAX edges incident to vertex sourceVertex
    FOR each vertex v adjacent to sourceVertex do{
        IF comp contains v {
            Update the distance of vertex stored in comp
        }
    }

    pimMin = untagged local vertex v in graph with minimum distance
    IF (pimMin NOT EQUAL to NULL) {
         pimMinDist = pimMin.distance
    }
    ELSE{
        pimMinDist = INFINITY
    }
    sourceVertex = pimMin

    remoteMin = untagged remote vertex v in graph with minimum distance
    IF (remoteMin NOT EQUAL to NULL) {
         remoteMinDist = remoteMin.distance
    }
    ELSE{
        remoteMinDist = INFINITY
    }

    compMin = untagged remote vertex v in graph with minimum distance
    IF (compMin NOT EQUAL to NULL) {
        compMinDist = compMin.distance
    }
    ELSE{
        compMinDist = INFINITY
```

```
    }

    IF( pimMinDist > compMinDist) {

        IF (pimMin is NOT NULL){
            FOR each local vertex v in graph do {
                IF (v is not tagged AND v.distance <= pimMinDist) {
                    ADD v to comp
                }
            }
        }
        IF ((pimMinDist >= remoteMinDist) OR (compMinDist >= remoteMinDist)) {
            FOR each remote vertex r in graph do {
                IF (r is not tagged AND r.distance <= pimMinDist) {
                    ADD r to comp
                }
            }
        }
        BREAK
    }
    ELSE{
        IF (pimMin != null) {
            IF ( pimMinDist  >  remoteMinDist) {
                ADD local and remote untagged vertices with distance <= compMinDist to comp
            }
            ELSE{
                IF( comp contains pimMin ) {
                    Remove pimMin from comp
                }
            }
        }
        ELSE{
            IF((remoteMinDist <= compMinDist)){
                Add remote untagged vertices with distance <= compMinDist to comp
            }
        }
    }
  }

  IF (comp  is not empty){
    compMin = vertex in comp with minimum distance
    sendParcel(pim, dataPayload, inputParcel.getContext(), remoteUuids);
  }
  sendParcel(pim, new Boolean(false), inputParcel.getContext());

}
```

## 5.3 Algorithm for Sub-Graph Isomorphism

Variables used ---
*mapping* - Hashtable mapping vertex in subgraph vertex in target graph
*possibleRemoteNodes* - Set containing subgraph vertices which may be mapped to remote vertices
*sortedSubgraphVertices* - sorted set of subgraph vertices; sorted in non-decreasing order of node degree
*possibleRemoteNodesMapping* - Hashtable mapping vertex to a set containing the remote nodes. A vertex may be mapped to one of the vertices in this set

public findSubgraph(SimPIM pim, SimParcel inputparcel){

*sortedSubgraphVertices* = SET of subgraph vertex Ids sorted in non-decreasing order of their degree
*startSubGraphNodeIndex* = index of the last element in *sortedSubgraphVertices* (a vertex with maximum degree)

WHILE (*startSubGraphNodeIndex* != -1) {

//traverse the subgraph vertices in non-increasing order of their degree
FOR each *vertexId* stored in *sortedSubgraphVertices* starting at *startSubGraphNodeIndex* and in reverse order do {

IF (*vertexId* is previously mapped to remote vertex) {
remove the mapping
backtrack to previous vertex
continue the FOR loop
}

IF (*vertexId* is in *possibleRemoteNodes* ){
remove it from *possibleRemoteNodes* and *possibleRemoteNodesMapping*
}

boolean *isMatched* = false;
IF (partial result contains probable mapping for *vertexId*){
*isMatched* = checkPartialMapping(*vertexId*)
}
ELSE{
*degree* = degree of subgraph vertex *vertexId* - 1
*isMatched* = markNode(*vertexId*, *degree*);
}

IF (not *isMatched*) {
backtrack to previous vertexId
}
ELSE {

```
            IF (vertexId mapped to local vertex) {
                Boolean isAdjMaintained = checkPossibleRemoteVerticesAdjacency(vertexId)
                IF (isAdjMaintained is TRUE) {
                    Advance to next vertex
                }
                ELSE {
                    Repeat the loop for the same element
                }
            }
        }
    }

    Hashtable mapping contains mapping of subgraph vertices to local graph vertices
    Add that to the partial result in the input parcel.

    LOOP through the SET possibleRemoteNodes and send the requests to appropriate PIM
CAGES on which their probable matching vertices reside locally
    reinitialiaze the data structures used.
    startSubGrapgNodeIndex = index of the vertex in subgraph to which next value can be
assigned
  }//while(startSubGraphNodeIndex != -1)

}// findSubgraph ... ends


private boolean checkPossibleRemoteVericesAdjacency(Integer vertexId){

    boolean isAdjMaintained = true;

    DO{
        FOR each vertex v with only one mapping vertex m in possibleRemoteNodesMapping {
            Check if adjacency is maintained by mapping vertex v to its probable mapping m
            IF (adjacency not maintained) {
                RETURN FALSE
            }
            Add pair (v, m) to mapping
            Remove v from possibleRemoteNodes and from
  possibleRemoteNodesMapping
            Tag m
        }

        FOR each vertex v adjacent to vertexId in subgraph do {
            IF (v is in possibleRemoteNodes){

                boolean isFinalRemoteNode = true;
                boolean allMatched = true;
```

```
                FOR each vertex r adjacent to v do {
                    IF (r is not in possibleRemoteNodes and finalRemoteNodes) {
                        isFinaRemoteNode = false
                        IF (r not in mapping) {
                            allMatched = FALSE
                        }
                    }
                }

                IF(allMatched){
                    IF(isFinalRemoteNode){
                        Remove v from possibleRemoteNodes
                        Add v to finalRemotedNodes
                    }
                    ELSE{
                        FOR each vertex rm in possible mapping for vertex v {
                            IF (rm is not tagged) {
                                Check if adjacency is maintained by mapping vertex v to vertex rm
                                IF adjacency maintained {
                                    Add pair (v, rm) to mapping
                                    Remove v from possibleRemoteNodes and from
    possibleRemoteNodesMapping
                                    Tag rm
                                    Update possibleRemoteNodesMapping to reflect the mapping
                                    BREAK
                                }
                            }
                        }
                    }

                    IF (allMatched AND no matching vertex found) {
                        RETURN TRUE
                    }
                }//if(allMatched)
            }
        }//for(int j = 0 ; j < adjNodes.size() ; j++)

    }WHILE there exists a vertex in possibeRemoteNodesMapping with only one probable
mapping

}//checkPossibleRemoteNodes.. end

private boolean markNode(Integer vertexId, int degree) {
    boolean isMarked = false;
    boolean preMapFound = false;
```

```
int premapNodeId = 0;
boolean areAllAdjNodesPossibleRemote = true;
boolean areAllAdjNodesMapped = true;
boolean isFinalRemoteNode = true;

FOR each vertex v adjacent to vertex vertexId {
    IF (v not in possibleRemoteNodes){
        areAllAdjNodesPossibleRemote = false
    }
    IF (v not in mapping && v not in finalRemoteNodes ) {
        areAllAdjNodesMapped = false
    }
    IF (v not in possibleRemoteNodes && v not in finalRemotedNodes) {
        isFinalRemoteNode = false; //not sure..
    }
}
//check if node to be added to finalRemoteNodes
IF (areAllAdjNodesMapped AND isFinalRemoteNode) {
    Add vertexId to finalRemotedNodes
    isMarked = TRUE
    RETURN isMarked
}
//if all adj nodes are possible remote then add node to possibleRemoteNodes
IF (areAllAdjNodesPossibleRemote) {
    Add vertexId to possibleRemoteNodes
    isMarked = TRUE
    RETURN isMarked
}
//if mapping contains nodeId, remove it
IF (vertex is matched already) {
    premapNodeId = matching of vertexId
    remove the matching of vertexId from mapping
    untag matched node
}

FOR each vertex v having degree equal to or greater than vertexId {

    IF (premapNodeId is not equal to ZERO){

        IF (v != premapNodeId){
            Continue FOR LOOP
        }
        premapNodeId = 0 ;

        IF v is the last candidate matching vertex for vertexId {
            IF (areAlladjNodesMapped){
```

```
                    isMarked = FALSE
                    RETURN isMarked
                }
                ELSE {
                    Find a SET of possible remote vertices to which vertexId can be mapped
maintaining the adjacency and add a pair (vertexId, SET of remote vertices) to
possibeRemoteNodesMapping
                    IF (only one possible remote node rId found) {
                        Add pair (vertexId, rId) to mapping
                        Tag the remote vertex
                        Update possibeRemoteNodesMapping by deleting the rId from the list of
probable remote vertices for the subgraph vertices in that table
                    }
                    ELSE {
                        Add vertexId to possibleRemoteNodes
                    }
                    isMarked = TRUE
                    RETURN isMarked
                }
            }
        }// if(premapNodeId != 0) .. end

        IF (v is not tagged) {
            Check if mapping vertexId to v preserves adjacency
            IF (adjacency is maintained) {
                isMarked = TRUE
                add a pair (vertexId, v) to mapping
                tag v
            }
        }
    }
}

IF (isMarked is FALSE ) {
    IF (areAllAdjNodesMapped) is FALSE) {
        Add vertexId to possibleRemoteNodes
        isMarked = TRUE
    }
}
RETURN isMarked
}// markNode.. ends
```